

Matching Logic

Grigore Rosu

University of Illinois at Urbana-Champaign


Joint work with Andrei Stefanescu and
Chucky Ellison. Started with Wolfram Schulte
at Microsoft Research in 2009

Question

... could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Hoare logic

$\{\pi_{\text{pre}}\} \text{code} \{\pi_{\text{post}}\}$



Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
 - Typically skipped: considered expensive and useless
- Model checkers for L
 - Based on some adhoc encodings/models of L
- Program verifiers for L
 - Based on some other adhoc encodings/models of L
- Runtime verifiers for L
 - Based on yet another adhoc encodings/models of L
- ...

Example of C Program

- What should the following program evaluate to?

```
int main(void) {  
    int x = 0;  
    return (x = 1) + (x = 2);  
}
```

- According to the C “standard”, it is **undefined**
- GCC4, MSVC: it returns **4**
GCC3, ICC, Clang: it returns **3**
By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**

A Formal Semantics Manifesto

- Programming languages must have formal semantics! (period)
 - And analysis/verification tools should build on them
- Informal manuals are not sufficient
 - Manuals typically have a formal syntax of the language (in an appendix)
 - Why not a formal semantics appendix as well?

Motivation and Goal

- We are facing a semantic chaos
 - Operational, denotational, axiomatic, etc.
 - Problematic when dealing with large languages
- Why so many semantic styles?
 - Since none of them is ideal, they have limitations
- We want a powerful, unified foundation for programming language semantics and verification
 - One semantics to serve all the purposes!

Minimal Requirements for an Ideal Language Semantic Framework

- Should be **expressive**
 - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be **executable**
 - So we can test it and use it in tools (symb. exec.)
- Should be **modular/compositional** (thus scale)
 - So each feature is defined once and for all
- Should serve as a **program logic**
 - So we can also prove programs correct with it

Current Semantic Approaches

-Structural Operational Semantics-

- Executable
- Not very modular/compositional (except MSOS)
- Not appropriate for verification
- Only interleaving semantics

Current Semantic Approaches

-Evaluation Contexts-

- Executable
- Modular, deals better with control
- Very syntactic, rigid
 - Enforces substitution-based definitions
 - Unsuitable for environment-based definitions
- Not appropriate for verification
- Only interleaving semantics

Current Semantic Approaches

-Denotational Semantics-

Reasonable trade-offs

- Mathematical model, somehow compositional
- Not very executable
 - Norish's C semantics is not executable
 - factorial(5) crashes Papaspyrou's C semantics
- Not very good for verification
- Poor for concurrency
- Requires expert knowledge

Current Semantic Approaches

-The Chemical Abstract Machine-

CHAM

- Intuitive computational model
- True concurrency (like in nature)
- No machine support
 - It would be very hard, because of its airlock
- Not appropriate for verification

Current Semantic Approaches

-Floyd-Hoare Logic-

- Good for program verification
- Requires encodings of structural program configuration properties as predicates
 - Heap, stacks, input/output, etc.
 - Framing is hard to deal with
- Not based on a formal executable semantics
 - Thus, hard to test
 - Semantic errors found by proving wrong properties
 - Soundness rarely or never proved in practice
- Implementations of Floyd-Hoare verifiers for real languages still an art, who few master

Towards a Better Semantic Approach

Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

- **Expressive**
 - Any logic can be represented in RL (it is reflective)
- **Executable**
 - Quite efficiently; Maude often outperforms SML
- **Modular**
 - Allows rules to only “match” what they need
- Can potentially serve as a **program logic**
 - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs

Rewriting Logic Semantics Project

- Project started jointly with Meseguer in 2003-4
- Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
- Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
 - But they still had their inherent limitations
- Appropriate techniques/methodologies needed

The K Framework

k-framework.org

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching undergraduate courses
- Main ideas:
 - Represent program configurations as a potentially **nested structure of cells** (like in the CHAM)
 - Flatten syntax into special **computational structures** (like in refocusing for evaluation contexts)
 - Define the semantics of each language construct by **semantic rules** (a small number, typically 1 or 2)

Complete K Definition of KernelC

```

MODULE KERNELC-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX  $Exp ::= Exp * Exp$  [strict]
|  $DeclId$ 
|  $Id$ 
|  $Int$ 
|  $Exp \rightarrow Exp$  [strict]
|  $Exp ++$ 
|  $Exp == Exp$  [strict]
|  $Exp != Exp$  [strict]
|  $Exp < Exp$  [strict]
|  $Exp \% Exp$  [strict]
|  $! Exp$ 
|  $Exp \&\& Exp$ 
|  $Exp ? Exp : Exp$ 
|  $Exp || Exp$ 
|  $printf("%d", Exp)$  [strict]
|  $scanf("%d", \& Exp)$ 
|  $scanf("%d", Exp)$  [strict]
|  $NULL$ 
|  $PointerId$ 
|  $(int*)malloc(Exp * sizeof(int))$  [strict]
|  $free(Exp)$  [strict]
|  $* Exp$  [strict]
|  $Exp [ Exp ]$ 
|  $Exp = Exp$  [strict2]
|  $Id ( List(Exp) )$  [strict2]
|  $Id ()$ 
|  $random()$ 
|  $srandom(Exp)$  [strict]
SYNTAX  $Stmt ::= Exp ;$  [strict]
|  $\{ \}$ 
|  $\{ StmtList \}$ 
|  $if(Exp) Stmt$ 
|  $if(Exp) Stmt \text{ else } Stmt$  [strict1]
|  $while(Exp) Stmt$ 
|  $return Exp ;$  [strict]
|  $DeclId List(DeclId) \{ StmtList \}$ 
|  $\#include< StmtList >$ 
SYNTAX  $StmtList ::= StmtList StmtList$ 
|  $Stmt$ 
SYNTAX  $Pgm ::= StmtList$ 
SYNTAX  $Id ::= main$ 
SYNTAX  $PointerId ::= *$   $PointerId$  [ditto]
|  $Id$ 
SYNTAX  $DeclId ::= int Exp$ 
|  $void PointerId$ 
SYNTAX  $StmtList ::= stdio.h$ 
|  $stdio.h$ 
SYNTAX  $List(Bottom) ::= List(Bottom) , List(Bottom)$  [assoc hybrid id: () strict]
|  $()$ 
|  $Bottom$ 
SYNTAX  $List(PointerId) ::= List(PointerId) , List(PointerId)$  [ditto]
|  $List(Bottom)$ 
|  $PointerId$ 
SYNTAX  $List(DeclId) ::= List(DeclId) , List(DeclId)$  [ditto]
|  $DeclId$ 
|  $List(Bottom)$ 
SYNTAX  $List(Exp) ::= List(Exp) , List(Exp)$  [ditto]
|  $Exp$ 
|  $List(DeclId)$ 
|  $List(PointerId)$ 
END MODULE

```

```

MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNELC-SYNTAX
MACRO  $I \ E = E ? 0 : I$ 

MACRO  $E_1 \&\& E_2 = E_1 ? E_2 : 0$ 

MACRO  $E_1 || E_2 = E_1 ? I : E_2$ 

MACRO  $if(E) St = if(E) St \text{ else } \{ \}$ 

MACRO  $NULL = 0$ 

MACRO  $I () = I () ()$ 

MACRO  $int * PointerId = int PointerId$ 

MACRO  $\#include< Stmts > = Stmts$ 

MACRO  $E_1 [ E_2 ] = E_1 + E_2$ 

MACRO  $scanf("%d", \& * E) = scanf("%d", E)$ 

MACRO  $int * PointerId = E = int PointerId = E$ 

MACRO  $int X = E ; = int X ; X = E ;$ 

MACRO  $stdio.h = \{ \}$ 

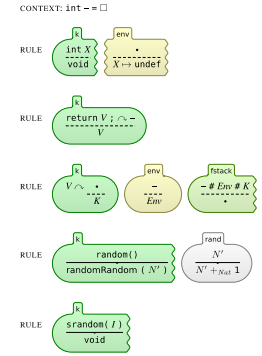
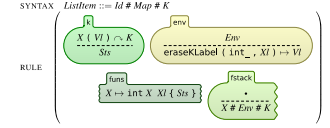
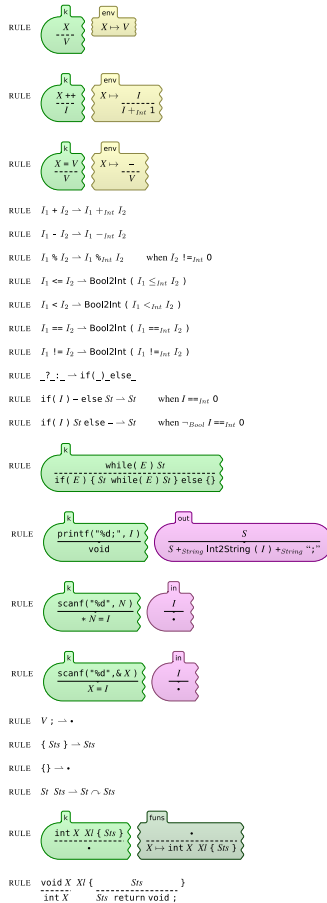
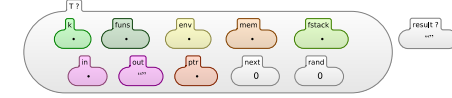
MACRO  $stdio.h = \{ \}$ 
END MODULE

```

```

MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K-KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:

```



CONTEXT: $* \square = -$

CONTEXT: $* \square ++$

SYNTAX $Val ::= Int$
| $void$

SYNTAX $Exp ::= Val$

SYNTAX $K ::= List(DeclId)$
| $List(Exp)$
| $List(PointerId)$
| Pgm
| $StmtList$
| $String$
| $restore(Map)$
| $undef$

SYNTAX $KResult ::= List(Val)$

SYNTAX $List(K) ::= Nat * Nat$

RULE $N_1 * N_2 \mapsto *$

RULE $N_1 * * Nat \ N \mapsto N , N_1 * * N$

SYNTAX $List(Val) ::= List(Val) , List(Val)$ [ditto]

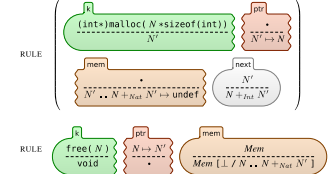
SYNTAX $List(Exp) ::= List(Val)$

END MODULE

```

MODULE KERNELC-SIMPLE-MALLOC
IMPORTS K
IMPORTS KERNELC-SEMANTICS

```



END MODULE

```

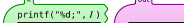
MODULE KERNEL-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX Exp ::= Exp + Exp [strict]
        | DeclId
        | Id
        | Int
        | Exp * Exp [strict]
        | Exp ++
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | Exp < Exp [strict]
        | Exp % Exp [strict]
        | Exp & Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp ? Exp : Exp
        | Exp || Exp
        | printf("sd", E, Exp) [strict]
        | scanf("sd", &E, Exp) [strict]
        | scanf("sd", Exp) [strict]
        | NULL
        | PointerId
        | (int+mallocl Exp + sizeof(int)) [strict]
        | tree(Exp) [strict]
        | * Exp [strict]
        | Exp < Exp [strict2]
        | Id ( List(Exp) ) [strict2]
        | Id ( )
        | random() Exp [strict]
SYNTAX Stmt ::= Exp; [strict]
        | { }
        | { StmtList }
        | if(Exp) Stmt
        | if(Exp) Stmt else Stmt [strict1]
        | while(Exp) Stmt
        | return Exp; [strict]
        | DeclId List(DeclId) { StmtList }
        | #include <Stms >
SYNTAX StmtList ::= StmtList StmtList
        | Stmt
SYNTAX Pgm ::= Stms
SYNTAX Id ::= main
SYNTAX PointerId ::= * PointerId [ditto]
        | Id
SYNTAX DeclId ::= int Exp
        | void PointerId
SYNTAX Stdio.h ::= stdio.h
SYNTAX List(Bottom) ::= List(Bottom) , List(Bottom) [assoc hybrid id: ( ) strict]
        | ( )
        | Bottom
SYNTAX List(PointerId) ::= List(PointerId) , List(PointerId) [ditto]
        | PointerId
SYNTAX List(DeclId) ::= List(DeclId) , List(DeclId) [ditto]
        | DeclId
        | List(Bottom)
SYNTAX List(Exp) ::= List(Exp) , List(Exp) [ditto]
        | Exp
        | List(DeclId)
        | List(PointerId)
END MODULE


MODULE KERNEL-DESUGARED-SYNTAX
IMPORTS K-LATEX
IMPORTS KERNEL-SYNTAX
MACRO ! E = E 7 0 : 1

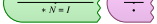
MACRO E1 && E2 = E1 7 E2 : 0
MACRO E1 || E2 = E1 1 1 : E2
MACRO if(E) S1 = if(E) S else { }
MACRO NULL = 0
MACRO I () = I ( )
MACRO int * PointerId = int PointerId
MACRO #include<Stms > = Stms
MACRO E1 [ E2 ] = * E1 + E2
MACRO scanf("sd", &E, E) = scanf("sd", E)
MACRO int * PointerId = E = int PointerId = E
MACRO int X = E ; = int X ; X = E ;
MACRO stdio.h = { }
MACRO stdlib.h = { }
END MODULE

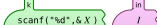
```

SYNTAX $Exp ::=$
 $\quad \mid \quad Exp = Exp$ [strict(2)]

RULE 

RULE 

RULE 


RULE 

RULE $V ; \rightarrow \cdot$

RULE $\{ S \} \rightarrow S \cdot$

RULE $\{ \} \rightarrow \cdot$

RULE $S \cdot S \rightarrow S \cdot \rightarrow S \cdot$

RULE 

RULE $\text{void } X \{ I \} \rightarrow S \cdot$

SYNTAX $KRval :: List[Val]$

SYNTAX $List[K] :: Nat \times Nat$

RULE $N_1 \dots N_i \rightarrow \cdot$

RULE $N_1 \dots \$Nat.N \rightarrow N \cdot N_1 \dots N$

SYNTAX $List[Val] :: List[Val], List[Val][dim0]$

SYNTAX $List[Exp] :: List[Val]$

END MODULE

```

    fun
      SumList
      String
      restore( Map )
    under
      KRval :: List[Val]
      List[K] :: Nat × Nat
    rule
      N1 .. Ni → ·
      N1 .. $Nat.N → N · N1 .. N
  
```

MODULE KERNELC-SIMPLE-MALLOC

IMPORTS K

IMPORTS KERNEL-SEMANTICS

RULE

RULE

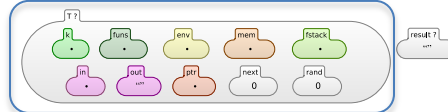
END MODULE

```

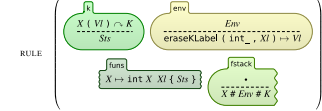
MODULE KERNEL-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT
SYNTAX Exp ::= Exp + Exp [strict]
          | Decid
          | Id
          | Int
          | Exp - Exp [strict]
          | Exp ++
          | Exp == Exp [strict]
          | Exp != Exp [strict]
          | Exp <= Exp [strict]
          | Exp < Exp [strict]
          | Exp % Exp [strict]
          | ! Exp
          | Exp && Exp
          | Exp ? Exp : Exp
          | Exp || Exp
          | print("vd", Exp) [strict]
          | scanf("vd", &Exp)
          | scanf("vd", Exp) [strict]
          | NULL
          | PointerId
          | (Int)
          | Tr

```

```
MODULE KERNELC-SEMANTICS
IMPORTS K-SHARED
IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
CONFIGURATION:
```



SYNTAX $ListItem ::= Id \# Map \# K$



CONTEXT: int - = □

RULE

$\frac{\text{int } X}{\text{void}}$	$\frac{\cdot}{X \mapsto \text{undef}}$
-------------------------------------	--

RULE $\text{return } V: \ominus =$

SYNTAX

SYNTAX

SYNTAX
SYNTAX

SYNTAX

SYNTAX

SYNTAX

SYNTAX

END MODU

MODULE K

MACRO

MACRO

MACRO

MACRO

MACRO

MACRO

```
MACRO scanf("%d",&*E) = scanf("%d",E)
```

```
MACRO int * PointerId = E = int PointerId = E
```

```
MACRO stdlib.h = {
```

END MODULE

$$\text{RULE } \frac{\text{void } X \text{ XI } \{ \quad \quad \quad Sfs \quad \quad \quad \}}{\text{int } X \quad \quad \quad Sfs \text{ return void ;}}$$

END MODULE

K Semantics are Useful

- Executable, help language designers
- Make teaching PL concepts hands-on and fun
- Currently compiled into
 - Maude, for execution, debugging, model checking
 - Latex, for human inspection and understanding
- Soon to be compiled to
 - OCAML, for fast execution
 - COQ, for meta-property verification

K Scales

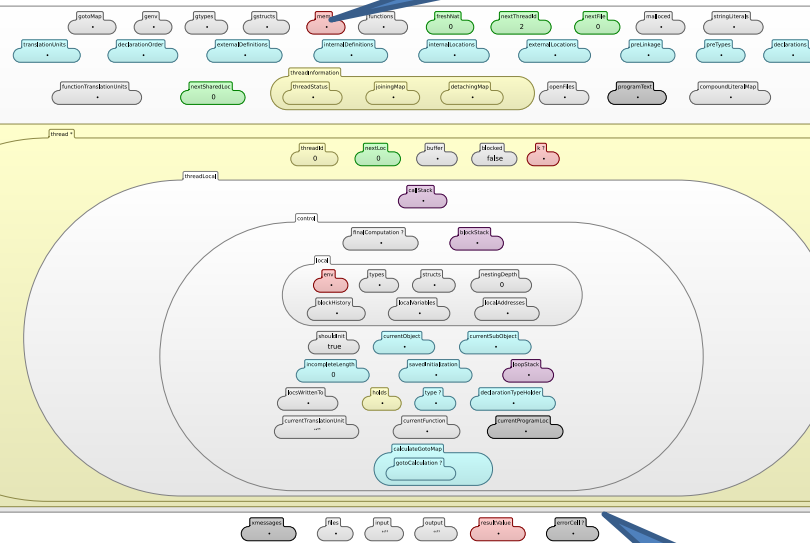
Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.

The K Configuration of C

Heap



75 Cells!

Statistics for the C definition

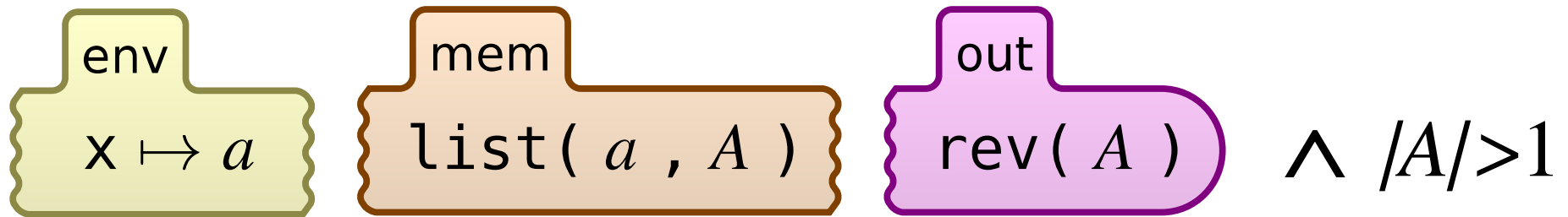
- Total number of rules: **~1200**
- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test, code from the obfuscated C competition, etc.)
 - Passed **99.2%** so far!
 - GCC 4.1.2 passes 99%, ICC 99.4%, Clang 98.3 (no opt.)
- *The most complete formal C semantics*
- Took more than 18 months to define ...
 - Wouldn't it be uneconomical to redefine it in each tool?

Matching Logic = K + FOL

- A logic for reasoning about configurations
- Formulae
 - FOL over configurations, called **patterns**
 - Configurations are allowed to contain variables
- Models
 - Ground configurations
- Satisfaction
 - **Matching** for configurations, plus FOL for the rest

Examples of Patterns

- x points to sequence A with $|A| > 1$, and the reversed sequence $\text{rev}(A)$ has been output



- **untrusted()** can only be called from **trusted()**



Matching Logic vs. Separation Logic

- Matching logic achieves separation through matching at the structural (term) level, not through special logical connectives (*)
- Matching logic realizes separation at all levels of the configuration, not only in the heap
 - the heap was only 1 out of the 75 cells in C's def.
- Matching logic can stay within FOL, while separation logic needs to extend FOL
 - Thus, we can use the existing SMT provers, etc.

Matching Logic as a Program Logic

- Hoare style - **not recommended**

$$\{\pi_{\text{pre}}\} \text{ code } \{\pi_{\text{post}}\}$$

– One has to redefine the PL semantics – **impractical**

- Rewriting (or K) style – **recommended**

$$\textit{left}[\text{code}] \rightarrow \textit{right}$$

– One can reuse existing K semantics – **very good**

Example – Swapping Values

$\$$ {

 void swap(int *x, int *y)

 {

 int t;

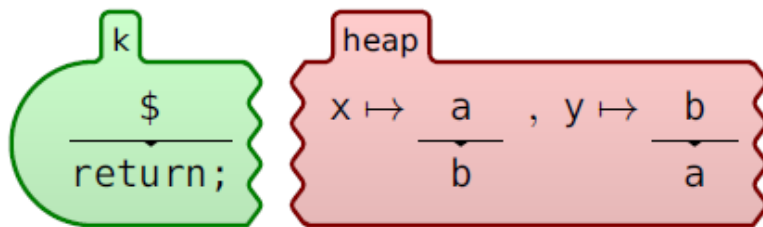
 t=*x;

 *x=*y;

 *y=t;

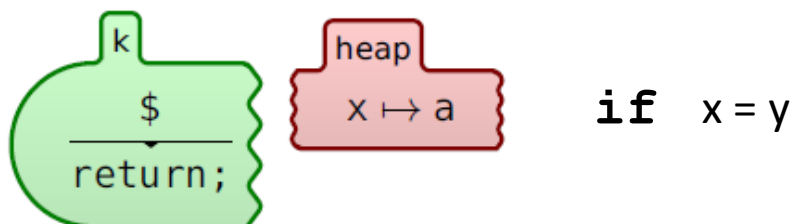
 }

- What is the K semantics of the swap function?
- Let $\$$ be its body



```

rule <k> $ => return; <_/k>
    <heap_>
        x|->(a=>b) ,
        y|->(b=>a)
    <_/heap>
    
```



```

rule <k> $ => return; <_/k>
    <heap_> x|-> a <_/heap>
    if x = y
    
```

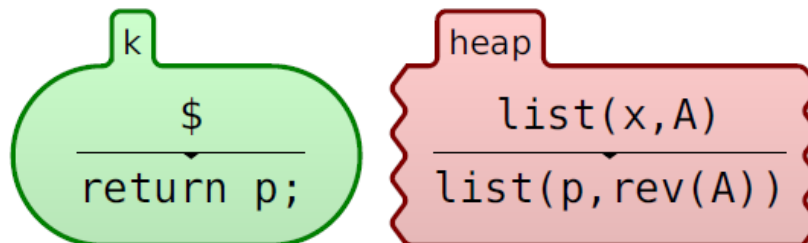
Example – Reversing a list

```

struct listNode* reverse(struct listNode *x)
{
    struct listNode *p;
    struct listNode *y;
    p = 0 ;
    while(x) {
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
    return p;
}

```

- What is the K semantics of the reverse function?
- Let \$ be its body



rule $\langle k \rangle \ \$ \Rightarrow \text{return } p; \ \langle /k \rangle$
 $\langle \text{heap_} \rangle \ \text{list}(x,A) \Rightarrow \text{list}(p, \text{rev}(A)) \ \langle _ / \text{heap} \rangle$

Partial Correctness

- We have two rewrite relations on configurations
 - given by the language K semantics; **safe**
 - given by specifications; **unsafe**, has to be proved
- Idea (simplified for deterministic languages):
 - Pick **left** → **right**. Show that always **left** → (**→** ∪ **→**)* **right** modulo matching logic reasoning (between rewrite steps)
- Theorem (soundness):
 - If **left** → **right** and “**config** matches **left**” such that **config** has a normal form for →, then “**nf(config)** matches **right**”

MatchC Tool DEMO

try it online first at

<http://fsl.cs.uiuc.edu/index.php/Special:MLOnline>

Semantic Execution

John Regehr and his team included the K semantics of C as part of his CSMITH tool chain, to make sure that the generated C programs are defined

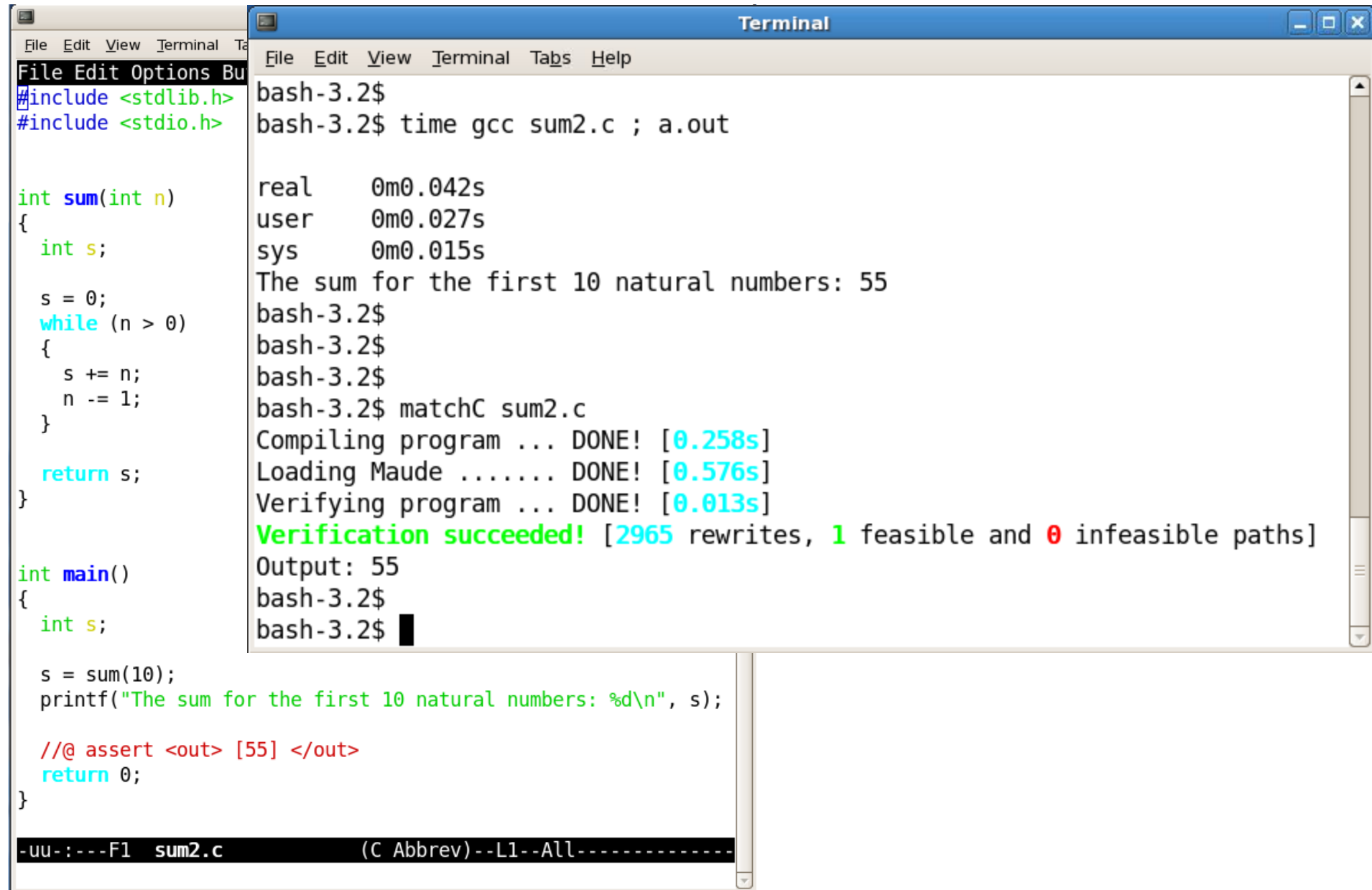
```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>

struct listNode {
    int val;
    struct listNode *next;
};

int main()
{
    struct listNode *x;
    x = (struct listNode *) malloc(sizeof(struct listNode));
    printf("%d\n", x->val);
    return 0;
}

bash-3.2$ time ./undefined.c
real    0m0.052s
user    0m0.022s
sys     0m0.018s
0
bash-3.2$
bash-3.2$
bash-3.2$ matchC undefined.c
Compiling program ... DONE! [0.207s]
Loading Maude ... DONE! [0.576s]
Verifying program ... DONE! [0.003s]
Verification failed! [174 rewrites, 0 feasible and 0 infeasible paths]
Output:
Generating error .... DONE! [0.155s]
Check undefined.kml for the complete output.
bash-3.2$
```

Assertion Checking



The image shows a code editor window on the left and a terminal window on the right. The code editor contains a C program named `sum2.c` that calculates the sum of the first 10 natural numbers. The terminal window shows the execution of the program, the compilation of the Maude model checker, and the successful verification of the program's assertion.

```
File Edit View Terminal Tabs Help
File Edit Options Bu
#include <stdlib.h>
#include <stdio.h>

int sum(int n)
{
    int s;

    s = 0;
    while (n > 0)
    {
        s += n;
        n -= 1;
    }

    return s;
}

int main()
{
    int s;

    s = sum(10);
    printf("The sum for the first 10 natural numbers: %d\n", s);

    //@ assert <out> [55] </out>
    return 0;
}

-uu-:---F1 sum2.c (C Abbrev)--L1--All-----
```

```
Terminal
File Edit View Terminal Tabs Help

bash-3.2$
bash-3.2$ time gcc sum2.c ; a.out

real    0m0.042s
user    0m0.027s
sys     0m0.015s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum2.c
Compiling program ... DONE! [0.258s]
Loading Maude ..... DONE! [0.576s]
Verifying program ... DONE! [0.013s]
Verification succeeded! [2965 rewrites, 1 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

Full Verification

```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
#include <stdlib.h>
#include <stdio.h>

int sum(int n)
//@ rule <k> $ => return (n * (n + 1)) / 2; </k> if n >= 0
{
    int s;

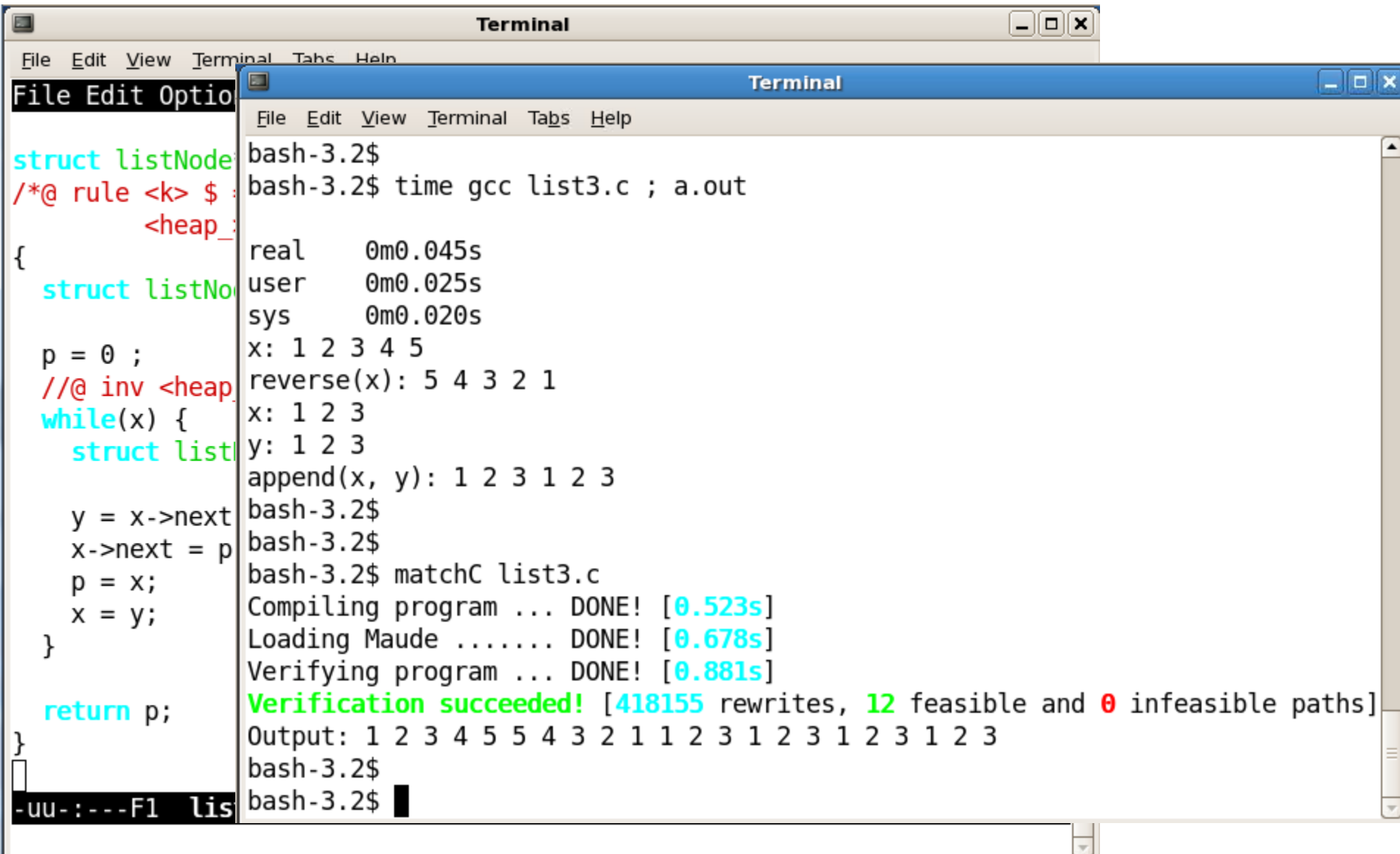
    s = 0;
    //@ inv s = ((old(r
    while (n > 0)
    {
        s += n;
        n -= 1;
    }

    return s;
}
```

```
Terminal
File Edit View Terminal Tabs Help
bash-3.2$
bash-3.2$ time gcc sum3.c ; a.out

real    0m0.042s
user    0m0.023s
sys     0m0.018s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum3.c
Compiling program ... DONE! [0.260s]
Loading Maude ..... DONE! [0.584s]
Verifying program ... DONE! [0.046s]
Verification succeeded! [33083 rewrites, 3 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

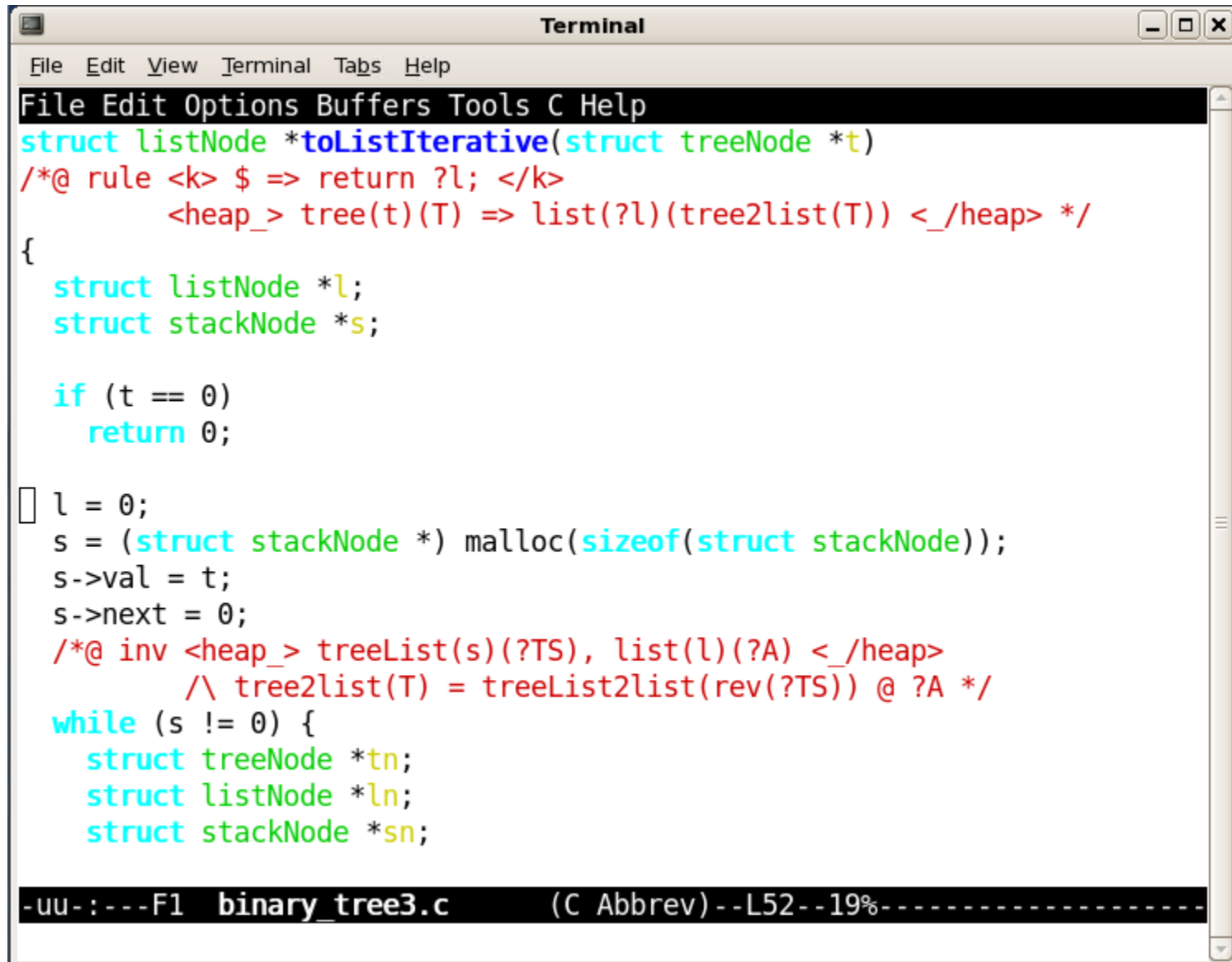
List Examples – Borrowed from SL tools



The image shows a terminal window with a blue title bar labeled "Terminal". The window contains the output of a Maude program execution. The program defines a list structure and performs a reversal operation. The output shows the execution time for compiling, loading, and verifying the program, followed by the verification result and the final list state.

```
bash-3.2$  
bash-3.2$ time gcc list3.c ; a.out  
real    0m0.045s  
user    0m0.025s  
sys     0m0.020s  
x: 1 2 3 4 5  
reverse(x): 5 4 3 2 1  
x: 1 2 3  
y: 1 2 3  
append(x, y): 1 2 3 1 2 3  
bash-3.2$  
bash-3.2$  
bash-3.2$ matchC list3.c  
Compiling program ... DONE! [0.523s]  
Loading Maude ..... DONE! [0.678s]  
Verifying program ... DONE! [0.881s]  
Verification succeeded! [418155 rewrites, 12 feasible and 0 infeasible paths]  
Output: 1 2 3 4 5 5 4 3 2 1 1 2 3 1 2 3 1 2 3 1 2 3  
bash-3.2$  
bash-3.2$
```


Beyond Separation Logic Tools



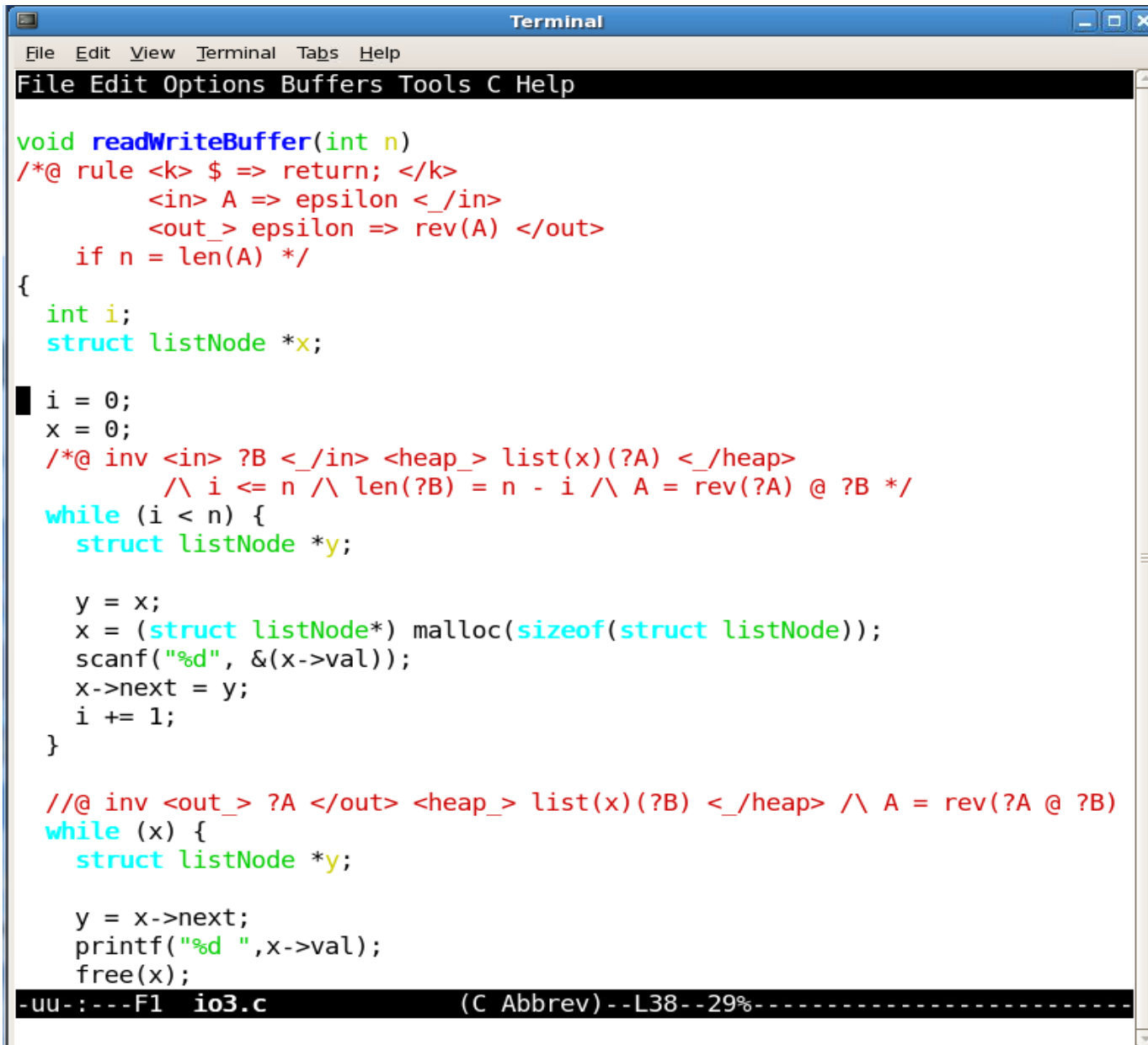
```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help
struct listNode *toListIterative(struct treeNode *t)
/*@ rule <k> $ => return ?l; </k>
    <heap_> tree(t)(T) => list(?l)(tree2list(T)) <_/heap> */
{
    struct listNode *l;
    struct stackNode *s;

    if (t == 0)
        return 0;

    l = 0;
    s = (struct stackNode *) malloc(sizeof(struct stackNode));
    s->val = t;
    s->next = 0;
    /*@ inv <heap_> treeList(s)(?TS), list(l)(?A) <_/heap>
        /\ tree2list(T) = treeList2list(rev(?TS)) @ ?A */
    while (s != 0) {
        struct treeNode *tn;
        struct listNode *ln;
        struct stackNode *sn;
```

-uu-:---F1 binary_tree3.c (C Abbrev)--L52--19%-----

Beyond Separation Logic – I/O



```
Terminal
File Edit View Terminal Tabs Help
File Edit Options Buffers Tools C Help

void readWriteBuffer(int n)
/*@ rule <k> $ => return; </k>
    <in> A => epsilon <_in>
    <out_> epsilon => rev(A) </out>
    if n = len(A) */
{
    int i;
    struct listNode *x;

    i = 0;
    x = 0;
    /*@ inv <in> ?B <_in> <heap_> list(x)(?A) <_heap>
        /\ i <= n /\ len(?B) = n - i /\ A = rev(?A) @ ?B */
    while (i < n) {
        struct listNode *y;

        y = x;
        x = (struct listNode*) malloc(sizeof(struct listNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }

    /*@ inv <out_> ?A </out> <heap_> list(x)(?B) <_heap> /\ A = rev(?A @ ?B)
    while (x) {
        struct listNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
```

-uu-:---F1 io3.c (C Abbrev) --L38--29%-----

Beyond Separation Logic – Stack Inspection

```
Terminal
File Edit Options Buffers Tools C Help

void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if n >= 10 \/ in(hd(ids(S)), {main, trusted}) */
{
    printf("%d ", n);
    untrusted(n);
    any(n);
    if (n)
        trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if in(trusted, ids(S)) */
{
    printf("%d ", -n);
    if (n)
        any(n - 1);
}

void any(int n)
{
    // untrusted(n);
    if(n > 10)
        // possible security violated if n < 10
        trusted(n - 1);
}

-uu:---F1 stack_inspection.c (C Abbrev)--L14--12%-----
```

Conclusions

- K (semantics) and Matching Logic (verification)
- Formal semantics is useful and practical!
- One can use an executable semantics of a language *as is* also for program verification
 - As opposed to redefining it as a Hoare logic
- Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools